COVER FEATURE

Warp Processing: Dynamic Translation of Binaries to FPGA Circuits

Frank Vahid, University of California, Riverside
Greg Stitt, University of Florida
Roman Lysecky, University of Arizona

Warp processing dynamically and transparently transforms an executing microprocessor's binary kernels into customized field-programmable gate array (FPGA) circuits, commonly resulting in 2X to 100X speedup over executing on microprocessors. A new architecture and set of dynamic CAD tools demonstrate warp processing's potential.

Software consists of bits downloaded into a prefabricated hardware device. Traditional microprocessor software bits represent sequential instructions to be executed by a programmable microprocessor. In contrast, field-programmable gate array software bits represent a circuit to be mapped onto an FPGA's configurable logic fabric. Both software types free developers from needing to design hardware. Instead, developers simply download bits into a prefabricated hardware device to implement a desired computation.

A computation might execute faster as a circuit on an FPGA than as sequential instructions on a microprocessor because a circuit allows concurrency, from the bit to the process level.[1] For example, a bit reversal implemented in a circuit requires only a single clock cycle but might require dozens of cycles when executed as logic/shift instructions on a microprocessor. An arithmetic-level computation involving 20 multiplications might require only two clock cycles if 10 multipliers are available on the FPGA, but it would require 20 cycles or more on a microprocessor. A process-level computation with 10 independent 100-cycle threads might require only 100 cycles if each thread is implemented as its own circuit, but it would require 1,000 cycles or more if sequenced on a single microprocessor.

Several commercial and research tools seek to compile popular microprocessor-oriented software programming languages (such as C, C++, and Java) to FPGAs. Many such FPGA circuit compilers use profiling to detect a program's kernels—that is, small regions in the program that account for most of the program's execution (following the well-known 90/10 rule)—and map those kernels to circuits on an FPGA, leaving the rest of the program to execute on a microprocessor.

Only a small group of expert developers has adopted such FPGA circuit-compilation tools. Key barriers to adoption include the difficulty of integrating such tools into established microprocessor software development flows and the nonconformance of such tools to the important standard binary concept that forms the basis of the architectures-tools-applications ecosystem in many computing domains.

Warp processing seeks to overcome these barriers by making FPGAs invisible to the software developer. In warp processing, a compute platform transparently performs FPGA circuit compilation as a program's binary executes on a microprocessor—that is, dynamically. Benjamin Levine and Herman Schmit's program acceleration work dynamically reconfigured functional units using statically created circuits.[2] Nathan Clark and his colleagues complemented statically determined program subgraphs with dynamic decisions of functional unit reconfiguration.[3] Warp processors are fully dynamic and generate entire coprocessing circuits beyond functional units.

 Published by the IEEE Computer Society 0018-9162/08/$25.00 © 2008 IEEE

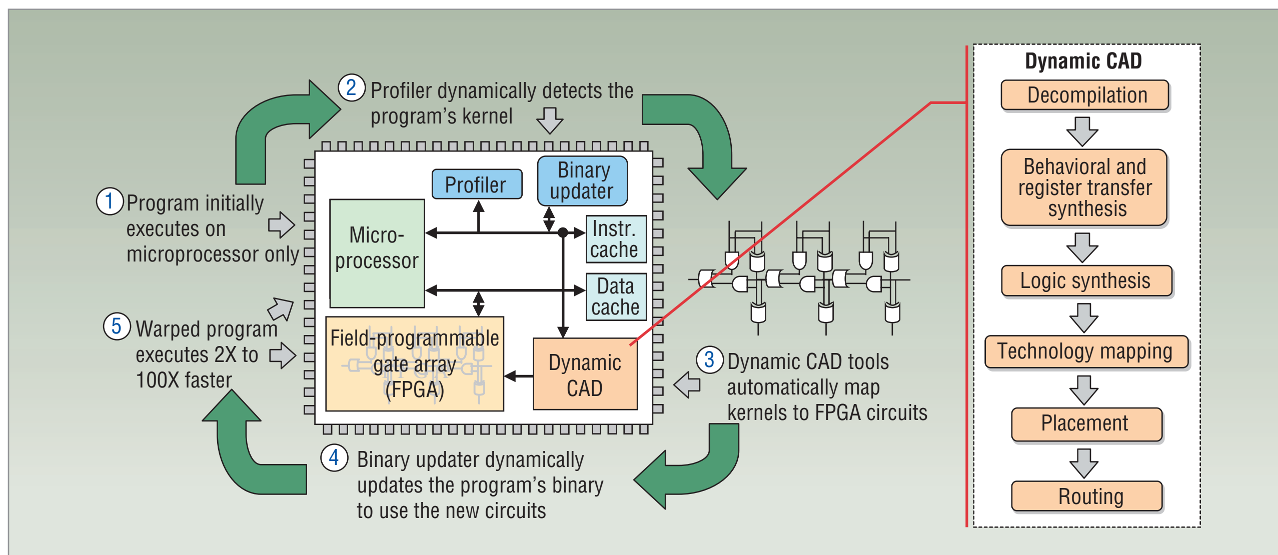

Figure 1. Warp processing overview. The profiler dynamically detects a downloaded program's binary kernels. The dynamic CAD tools map the kernels to FPGA circuits, and the binary updater dynamically updates the binary to use the new circuits. The net result can be dramatically faster—warped—execution.

WARP PROCESSING

Figure 1 provides an overview of warp processing. The architecture consists of a microprocessor and FPGA sharing instruction and data caches (or memory), a profiler, and dynamic CAD tools. A developer or end user initially downloads a program as microprocessor software (that is, a microprocessor binary). The profiler dynamically detects the binary's kernels, the dynamic CAD tools automatically map those kernels to FPGA circuits, and the binary updater dynamically updates the program's binary to use the new circuits. When the update takes place, the program's execution might suddenly speed up by a factor of 2, 10, or even more—in other words, the execution time "warps."

Profiling an executing binary is a widely investigated problem with numerous approaches that trade off accuracy and performance overhead.[4] Researchers have also developed solid solutions for dynamic binary updating.[5] Thus, developing effective dynamic CAD tools represents the main outstanding problem in enabling warp processing. Two key challenges in developing warp processing's dynamic CAD tools are compiling fast and efficient circuits from binary code instead of source code and quickly synthesizing a computation into an FPGA circuit using only lean dynamic-processing resources instead of powerful desktop workstations.

Decompilation

Warp processors synthesize circuits from executing binary code rather than from source code. However, binary code lacks high-level constructs (such as loops, arrays, and functions), which are readily detected in source-level code. Without such high-level constructs, synthesis from binaries might yield slower or bigger circuits. We use aggressive decompilation to address the challenge of synthesizing fast efficient circuits from binary code.

Decompilation involves recovering high-level constructs from binary code. Fortunately, researchers have developed sophisticated decompilation techniques for retargeting binaries from one microprocessor to another. These techniques can recover various if-then-else constructs, loops (including nested loops), arrays, functions, and more.[6]

However, efficient circuit compilation also requires two new decompilation techniques.[7] *Loop rerolling* detects an unrolled loop in a binary and replaces the code with a rerolled loop, thus letting a circuit synthesizer unroll the loop by an amount that matches available FPGA resources. Previous decompilation techniques also use loops to detect arrays, and synthesizers need arrays to effectively use FPGA smart buffers, which increase data reuse and thus decrease time-consuming memory accesses.[8] Rerolling also reduces control-flow graph size, thus significantly reducing the time for circuit synthesis, which typically uses superlinear (such as quadratic) algorithms with respect to the graph size.

The other new technique, *operator strength promotion*, detects strength-reduced operations (for example, a multiplication replaced by shifts and adds) and replaces them with stronger operators (for example, a multiplication), thus letting a circuit compiler use fast functional units (such as a multiplier) if available on the FPGA.

We developed a new decompilation tool, consisting of 15,000 lines of C code that incorporates key existing techniques and our two new techniques. The tool's output is a control/dataflow graph, which synthesis tools can convert to a fast and efficient circuit.

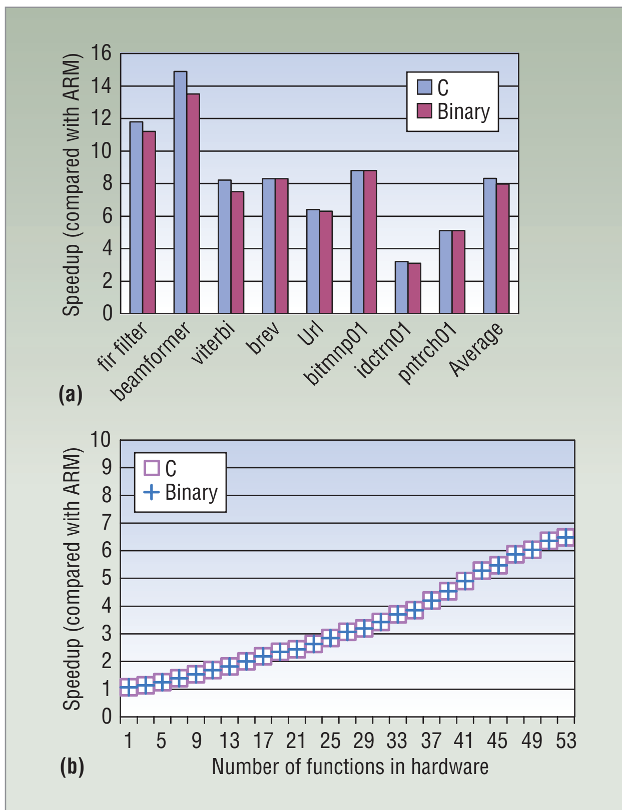

Figure 2. Comparison of FPGA circuit synthesis from C code and from decompiled binary code. (a) Various standard benchmark applications, showing only a 2.5 percent difference, and (b) in-depth study of the most frequent 53 functions of an H.264 decoder application, showing almost no difference.

We compared performance speedups (versus microprocessor-only execution) achieved when synthesizing kernels to FPGA circuits directly from C code to synthesizing from a decompiled control/dataflow graph generated by our decompilation tool.[9] Figure 2a gives results for various small (several hundred lines) embedded system benchmarks, showing nearly identical performance with using C. Speedups were nearly identical, being 8X faster on average than microprocessor execution.

Without our two new decompilation techniques, the binary approach would have yielded 33 percent less average speedup, with a worst case of 65 percent less. Without any decompilation, the binary approach actually yielded an average slowdown (not speedup) of 4X.

We also conducted an in-depth study on a large 16,000-line highly optimized H.264 video decoder application obtained through collaboration with Freescale.[10] Figure 2b shows that synthesis from binaries was nearly indistinguishable from synthesis from C.

Even used statically, synthesis from binaries supports numerous source-programming languages, leading to commercial products from Binachip and Critical Blue for static binary synthesis.

Dynamic CAD

To quickly convert a computational kernel into an FPGA circuit using only lean compute resources, we developed a complete suite of efficient CAD algorithms and a custom FPGA fabric intended to enable such efficient CAD tasks.

FPGA CAD tasks, shown in Figure 1, include

- decompilation,
- behavioral synthesis (converting a control/dataflow graph to a data path and register transfers),
- register transfer synthesis (converting register transfers to logic),
- logic synthesis (minimizing logic),
- technology mapping (mapping logic to FPGA-compatible resources),
- placement (placing logic/compute resources within specific FPGA resources), and
- routing (creating connections between logic/compute resources).

Figure 3a shows average CAD task runtime and memory usage when converting Embedded Microprocessor Benchmark Consortium (EEMBC) application kernels to FPGA circuits, using Xilinx ISE running on a powerful 3.2-GHz Pentium D-based desktop workstation. The figure doesn't show data for behavioral synthesis and register transfer synthesis because these tasks require orders of magnitude less time and memory than the others.

Routing is the most compute- and memory-intensive FPGA CAD task. Typical routing tool approaches iteratively reroute a circuit until the tool determines a valid or sufficiently optimized routing. Such approaches represent the FPGA's programmable elements using a large routing resource graph, consisting of nodes that correspond to every configurable switch within the FPGA (of which there might be hundreds of thousands). During each routing pass, the routing algorithms must search through and update the routing resource graph, requiring long execution times and much memory.

While building on such algorithms, we reduced execution time and memory use by developing a fast lean routing algorithm and designing a CAD-oriented FPGA fabric.[11] As Figure 4 shows, the fabric directly connects the configurable logic blocks' inputs and outputs to the switch matrices that handle routing. Then, instead of representing all configurable switches within the FPGA, our routing approach only needs to represent the larger switch matrices (each switch matrix consists of hundreds of configurable switches), significantly reducing the routing resource graph's memory requirements and reducing execution time required to search the graph during routing.

Our router is 10X faster and uses 20X less memory than the popular VPR routing algorithm. The tradeoff is a 30 percent reduction in maximum circuit execu-

tion speed and 10 percent more routing resource usage.

Our FPGA fabric also includes dedicated hard-core components, including multiply accumulators, data address generators, and loop control hardware, specifically designed to efficiently speed up microprocessor kernels, obtaining improvements roughly equal to the routing approach's overhead.

We verified our custom CAD-oriented FPGA design's functionality and performance through postlayout simulation targeting a 0.13-μm technology as part of the Intel Research Shuttle.

We also developed lean logic-synthesis, technology-mapping, and placement algorithms.[9] Typically, such CAD algorithms optimize the circuit implementation during each iteration. Our algorithms use single-pass optimizations, requiring orders of magnitude less memory and execution time than traditional approaches. The core of our logic-synthesis algorithm is an efficient two-level logic minimizer that's 15X faster and uses 3X less memory than Espresso-II. The tradeoff here is a 2 percent increase in circuit size.

Our technology mapper uses a hierarchical bottom-up graph-clustering algorithm that's 25X faster than commercial technology-mapping algorithms but only minimally impacts circuit delay. Our dependency-based positional-placement algorithm requires orders of magnitude less memory and execution than popular commercial and research placement tools, but with tradeoffs directly related to our routing algorithms' circuit performance. We obtained these efficiencies by focusing on microprocessor kernel speedup and by giving up some circuit performance for CAD efficiency.

The collection of lean FPGA CAD algorithms forms the Riverside Dynamic CAD tools. The RDCAD tools consist of 30,000 lines of C code. Figure 3b shows runtime and memory use for each RDCAD task on the 3.2-GHz desktop workstation. Runtimes are in fractions of seconds rather than tens of seconds to minutes, and memory use is only 3.6 Mbytes. RDCAD ran on a small low-cost embedded processor (a 40-MHz ARM7) in only 1.2 seconds using only 3.6 Mbytes of memory.

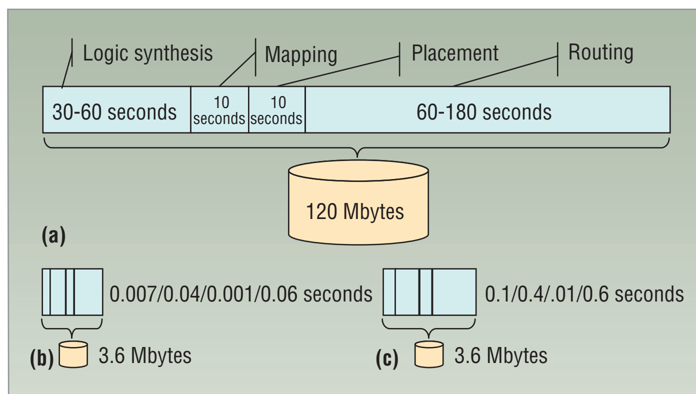

Figure 3. Typical time spent on CAD tasks by (a) a commercial FPGA CAD tool running on a desktop workstation, (b) the Riverside Dynamic CAD tools on the same workstation, and (c) the RDCAD tools on a lean 40-MHz ARM7 processor. Note: Not drawn to scale.

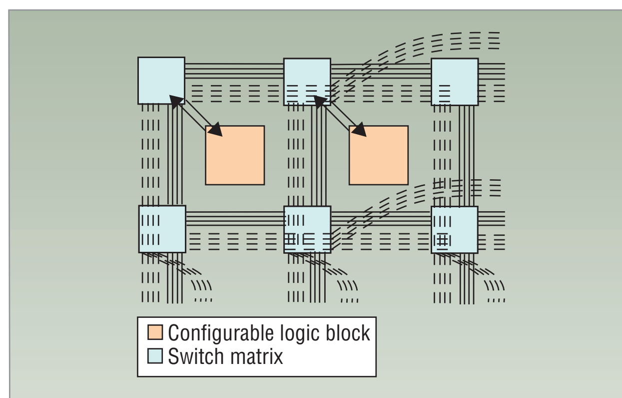

Figure 4. In the CAD-oriented FPGA, the configurable logic block inputs and outputs are directly connected to the switch matrices.

WARP PROCESSING SCENARIOS

Researchers can apply warp processing in two scenarios, depending on application runtime. Figure 5a shows the execution of a short-running application, in which the dynamic CAD tools run longer than the application. In this scenario, warp processing achieves no speedup for the first few executions, but warps future executions by saving and then reusing the application's saved FPGA configuration.

Figure 5b illustrates warp processing for longer-running applications requiring hours or days, such as in scientific computing. In this scenario, profiling and dynamic CAD finish well before the end of the application's first execution, allowing for warped execution of the remainder of the application. This scenario requires no saving of the FPGA configuration beyond an application's single execution, although the application could still use saved configurations for future executions.

RESULTS

We conducted various experiments to determine overall application speedups obtained by warp processing. We considered single-threaded applications as well as increasingly common multithreaded applications.

Single-threaded applications

We ran experiments on numerous single-threaded benchmark applications from various benchmark suites, including Powerstone, EEMBC, and MediaBench (see Table 1 for a list of these applications). We only considered applications amenable to speedup using FPGAs,

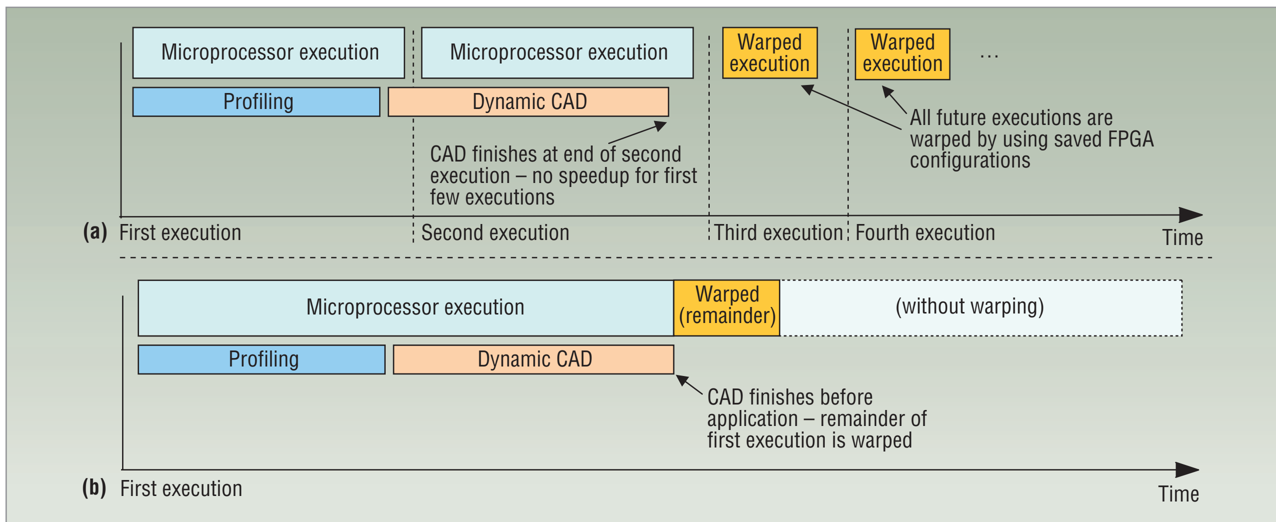

Figure 5. Warp processing scenarios: (a) repeated application warping, in which a short-running application is warped after several executions; and (b) one-time warping, in which a long-running application is warped during a single execution.

Table 1. Overview of benchmark applications.

Benchmark	Benchmark suite	Description
brev	Powerstone	Bit reversal
g3fax	Powerstone	Group three fax decode
matmul	Powerstone	Matrix multiplication
mpeg2	MediaBench	MPEG-2 decoder
pktflow	EEMBC	IP header validation
bitmnp	EEMBC	Bit manipulation
canrdr	EEMBC	Controller area network (CAN)
tblook	EEMBC	Table lookup and interpolation
ttsprk	EEMBC	Engine spark controller
matrix	EEMBC	Matrix operations
idct	EEMBC	Inverse discrete cosine transform
fir	EEMBC	Finite impulse response filter
rocm	Warp	RDCAD logic minimizer
prewitt	Warp (multithreaded, or MT)	Prewitt edge detection
search	Warp (MT)	Parallel search
moravec	Warp (MT)	Moravec image processing
wavelet	Warp (MT)	Wavelet transform
maxfilter	Warp (MT)	Maximum window image filter
N-body	Warp (MT)	Barnes-Hut N-body simulation

whose critical regions don't use floating-point arithmetic, dynamic memory allocation, recursion, or pointers (other than for array accesses), though advances in FPGA synthesis increasingly support such features. For other applications, warp processing would provide little or no speedup unless we rewrote them or developed new decompilation techniques. However, warp processing should never result in a slowdown. If warp processing can't speed up an application, the binary updater simply leaves the binary to execute on the microprocessor alone.

The warp-processing architecture simulated in these experiments uses an ARM9 operating at 200 MHz for the main microprocessor. All hardware regions execute in the FPGA at 100 MHz (some benchmarks could have executed at a faster frequency, but we held the frequency at 100 MHz for simplicity).

Such a 2-to-1 clock frequency ratio between microprocessor and FPGA is representative of various commercially available single-chip microprocessor/FPGA devices. The reported speedups hold for existing and future systems with similar frequency ratios, such as systems with 800-MHz microprocessors and 400-MHz FPGAs.

Our present warp FPGA fabric supports approximately 50,000 equivalent logic gates, roughly equal in logic capacity to a small Xilinx Spartan3 (XC3S50) FPGA. Warp processing required only 3,500 logic gates on average per application, with the largest cases being pktflow and ttsprk, which required 10,000 logic gates each. In 0.18-μm technology, our 50,000-gate FPGA fabric occupies roughly the same area as one ARM9 processor with a 32-Kbyte cache, or as a 64-Kbyte cache alone.

For comparison, modern commercial FPGAs have capacities of tens of millions of gates, and commercial single-chip microprocessor/FPGA devices allocate about 5X more silicon area to the FPGA than to the microprocessors and caches. The profiler required a small cache of several dozen entries and 2,300 logic gates. The dynamic CAD tools used an additional small, inexpensive ARM7 processor operating at 40 MHz.

The current architecture implements communication between the microprocessor and FPGA using a combination of shared memory, memory-mapped communication, and interrupts. The FPGA uses data-address generators, similar to digital signal processors (DSPs), to stream data required by FPGA circuits from memory. The microprocessor uses memory-mapped communication to initialize and enable the FPGA; it uses interrupts to detect a hardware circuit's completion.

We ran each application on an instruction-set simulator to obtain cycle counts for microprocessor execution. We ran application kernel binaries through our RDCAD tools to obtain kernel cycle counts after warp processing and to ensure that we obtained a minimum 100-MHz frequency. We inserted the necessary communication for data transfers between the microprocessor and FPGA (as determined by RDCAD tools) and counted those cycles too. A single data transfer between the microprocessor and FPGA required at least one cycle but at most two cycles.

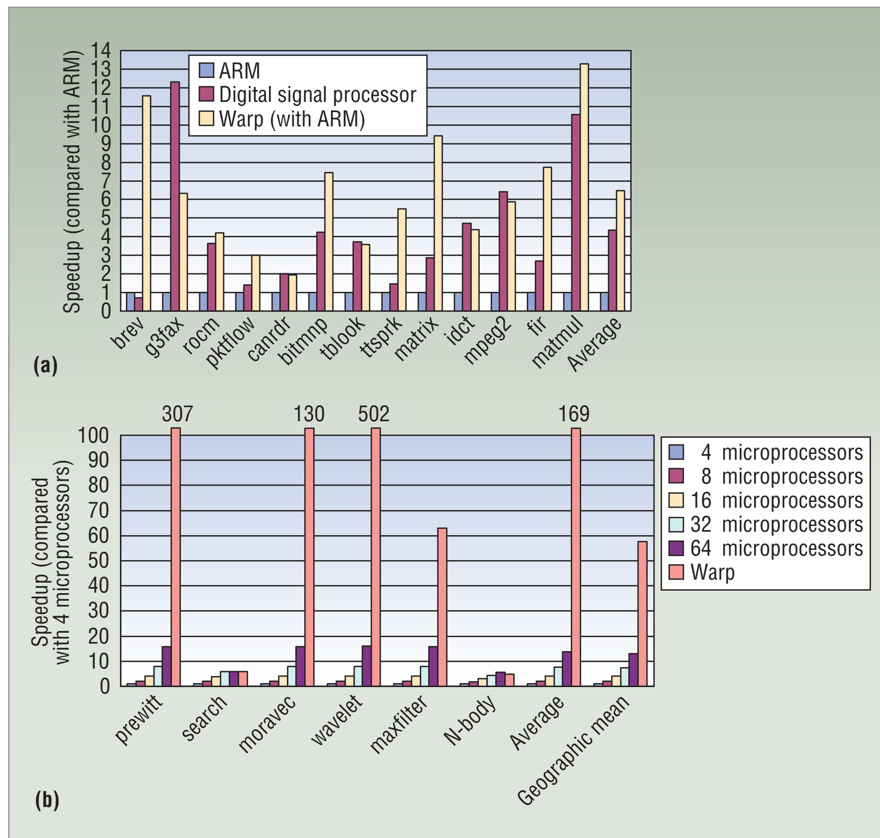

Figure 6. Speedup comparison. (a) Comparison of software execution on a digital signal processor (DSP) and warped execution on a warp processor to a 200-MHz ARM9 on single-threaded applications. (b) Comparison of multithreaded application speedups on various 400-MHz ARM11-based multiprocessors and warp processors.

Figure 6a compares the execution time of warp processing with microprocessor-only execution to which the data is normalized. As the figure shows, warp processing achieved an average application speedup of 6.5X, and speedup as high as 13.3X for matmul.

Figure 6a also shows speedups of a DSP—a TriMedia processor running at 200 MHz—versus the ARM9 processor. Like warp processing, the DSP exploits arithmetic-level parallelism to improve performance, but does so using a very large instruction word (VLIW) architecture. The DSP averaged 4.4X speedup compared to the ARM9, less than the 6.5X speedup of warp processing.

Warp processing was usually faster. The DSP was 2X faster for one benchmark and a few percent faster for some others. Warp processing gains versus the DSP came primarily from warp processing's ability to exploit more arithmetic-level parallelism (DSPs typically can execute only several operations in parallel) and to support a wider range of parallelism beyond arithmetic-level parallelism. The DSP outperformed warp processing when the application exhibited little parallelism, such that the DSP's faster clock frequency led to faster overall performance.

We also applied warp processing to SPEC desktop application benchmarks but found little speedup. Warp processing would have had to speed up tens or hundreds of large loops to achieve benchmark speedups, requiring very large FPGAs. Furthermore, many benchmarks used constructs, such as pointers, recursion, and dynamic memory allocation, that prevented circuit speedups.

On average, the dynamic CAD tools executed for 1.2 seconds. Thus, most of the embedded applications considered would require a saved FPGA configuration. For example, g3fax performs a group-three fax decoding for a single fax transmission. However, the dynamic CAD tools wouldn't have warped the execution until after the first fax was decoded. By saving the synthesized circuit, future fax transmissions would benefit from warp processing, providing faster fax decoding.

Some applications would benefit immediately from warp processing. For MPEG-2, the dynamic CAD tools would have completed after decoding only a few video frames, providing smoother video playback for the remainder of the video.

Multithreaded applications

Warp processing's benefits were most apparent for applications with much concurrency, such as brev (see Figure 6a), which has much bit-level concurrency, and matrix and fir, which have much arithmetic-level concurrency. We also examined multithreaded applications, which obviously have much thread-level concurrency.

Extending warp processing for multithreaded applications[12] required additional CAD tools and operating system support. A warp-aware operating system requests custom processors from the dynamic CAD tools for frequent threads. The CAD tools determine which and how many threads to synthesize. Memory access synchronization determines shared memory locations and synchronizes the execution of threads that share memory to reduce the number of needed direct memory access channels. After creating thread-accelerator circuits, the warp-aware operating system schedules threads onto both microprocessors and custom circuits.

Figure 6b shows warp-processing results for multithreaded applications that we developed, including multithreaded versions of image-processing and scientific-computing applications. Compared to a four-processor 400-MHz ARM11 system, warp processing obtained average speedups of 169X. Although much of the speedup came from executing threads in parallel, the speedups compared even with a 64-processor system illustrate that other factors, including arithmetic-level parallelism within threads, and custom communication, were significant. The size of the FPGA used equaled 36 ARM11s.

We are currently focusing on desktop, server, and scientific-computing applications. Initial results using high-end processors and high-end FPGAs demonstrate similar speedups for various applications.

Our warp processing work shows the technique's feasibility and potential, opening the door to new challenges, including dynamically allocating FPGA resources among multiple tasks, improving decompilation and synthesis to expand the applications that can be sped up, determining when to activate or terminate the dynamic CAD tools, synthesizing alternative accelerators that trade off performance and size, and reducing power or energy via warp processing. ■

References

1. K. Compton and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software," *ACM Computing Surveys*, vol. 34, no. 2, 2002, pp. 171-210.
2. B.A. Levine and H.H. Schmit, "Efficient Application Representation for HASTE: Hybrid Architectures with a Single, Transformable Executable," *Proc. Symp. FPGAs for Custom Computing Machines* (FCCM 03), IEEE Press, 2003, pp. 101-110.
3. N. Clark et al., "An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors," *Proc. Int'l Symp. Computer Architecture* (ISCA 05), IEEE Press, 2005, pp. 272-283.
4. M. Gschwind et al., "Dynamic and Transparent Binary Translation," *Computer*, Mar. 2000, pp. 54-59.
5. M.D. Hill et al., "Wisconsin Architectural Research Tool Set," *ACM SIGARCH Computer Architecture News*, vol. 21, no. 4, 1993, pp. 8-10.
6. C. Cifuentes and M. Van Emmerik, "UQBT: Adaptable Binary Translation at Low Cost," *Computer*, Mar. 2000, pp. 60-66.
7. G. Stitt and F. Vahid, "Binary Synthesis," *ACM Trans. Design Automation of Electronic Systems*, vol. 12, no. 3, 2007, pp. 1-30.
8. Z. Guo, A.B. Buyukkurt, and W. Najjar, "Input Data Reuse in Compiling Window Operations onto Reconfigurable Hardware," *Proc. Symp. Languages, Compilers, and Tools for Embedded Systems* (LCTES 04), ACM Press, 2004, pp. 249-256.
9. R. Lysecky, G. Stitt, and F. Vahid, "Warp Processors," *ACM Trans. Design Automation of Electronic Systems*, vol. 11, no. 3, 2006, pp. 659-681.
10. G. Stitt et al., "Hardware/Software Partitioning of Software Binaries: A Case Study of H.264 Decode," *Proc. Int'l Conf. Hardware/Software Codesign and System Synthesis* (CODES/ISSS 05), ACM Press, 2005, pp. 285-290.
11. R. Lysecky, F. Vahid, and S. Tan, "Dynamic FPGA Routing for Just-in-Time Compilation," *Proc. IEEE/ACM Design Automation Conf.* (DAC 04), ACM Press, 2004, pp. 954-959.
12. G. Stitt and F. Vahid, "Thread Warping: A Framework for Dynamic Synthesis of Thread Accelerators," *Proc. Int'l Conf. Hardware/Software Codesign and System Synthesis* (CODES/ISSS 07), ACM Press, 2007, pp. 93-98.

***Frank Vahid** is a professor at the University of California, Riverside. His research interests include embedded system programming and design. Vahid received a PhD in computer science from UC Irvine. He is a senior member of the IEEE and a member of the ACM. Contact him at vahid@cs.ucr.edu.*

***Greg Stitt** is an assistant professor at the University of Florida. His research interests include reconfigurable computing, embedded systems, and computer-aided design. Stitt received a PhD in computer science from UC Riverside. He is a member of the IEEE and the ACM. Contact him at gstitt@ece.ufl.edu.*

***Roman Lysecky** is an assistant professor at the University of Arizona. His research interests include embedded systems design, dynamic adaptability, hardware/software partitioning, field-programmable gate arrays, and low-power design methodologies. Lysecky received a PhD in computer science from UC Riverside. He is a member of the IEEE and the ACM. Contact him at rlysecky@ece.arizona.edu.*